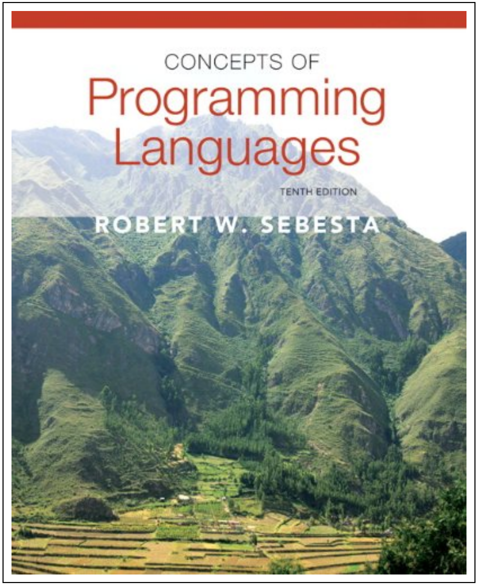
Concepts of Programming Languages

Lecture 1 - Introduction

Textbook


CONCEPTS OF
Programming Languages
TENTH EDITION
ROBERT W. SEBESTA

Administrivia

Reading:

Chapter 1

A good programming language is a conceptual universe for thinking about programming.

A. Perlis

Reasons for Studying Programming Languages

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Better use of languages that are already known
- Overall advancement of computing

Principles

Programming languages have four properties:

- Syntax
- Names
- Types
- Semantics

For any language:

- Its designers must define these properties
- Its programmers must master these properties

Syntax

Definition

The ***syntax*** of a programming language is a precise description of all its grammatically correct programs.

When studying syntax, we ask questions like:

- What is the grammar for the language?
- What is the basic vocabulary?
- How are syntax errors detected?

Names

Various kinds of entities in a program have names:

- variables
- types
- functions
- parameters
- classes
- objects

Names

Various kinds of entities in a program have names:

- variables
- types
- functions
- parameters
- classes
- objects

Named entities are bound in a running program to:

- Scope
- Visibility
- Type
- Lifetime

Types

Definition

A ***type*** is a collection of values and a collection of operations on those values.

Simple types:

- numbers
- characters
- booleans, . . . etc.

Structured types

- Strings,
- lists,
- trees,
- hash tables, . . . etc..

Definition

A language's ***type system*** can help to:

- Determine legal operations
- Detect type errors

Semantics

Definition

The meaning of a program is called its ***semantics***.

In studying semantics, we ask questions like:

- When a program is running, what happens to the values of the variables?
- What does each statement mean?
- What underlying model governs run-time behavior, such as function call?
- How are objects allocated to memory at run-time?

Paradigms

Definition

A programming ***paradigm*** is a pattern of problem-solving thought that underlies a particular genre of programs and languages.

There are four main programming paradigms:

- Imperative
- Object-oriented
- Functional
- Logic

Imperative Paradigm

Follows the classic von Neumann-Eckert model:

- Program and data are indistinguishable in memory
- Program = a sequence of commands
- State = values of all variables when program runs
- Large programs use procedural abstraction

Example imperative languages:

- Cobol
- Fortran
- C
- Ada
- Perl

The von Neumann-Eckert Model

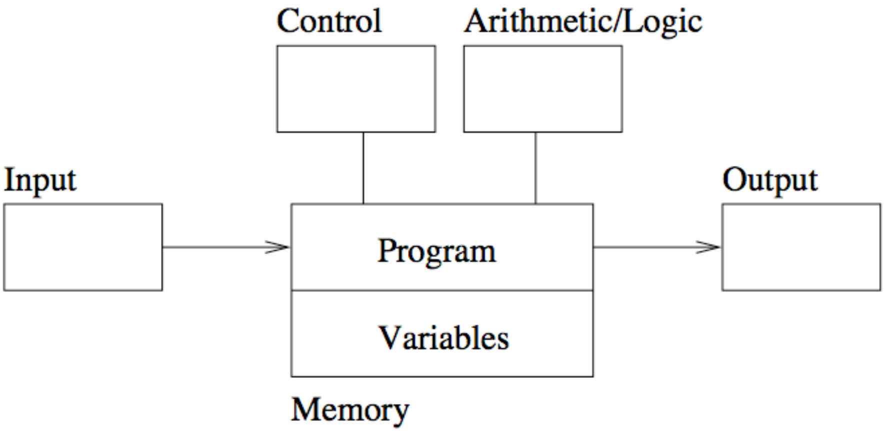


Figure 1.1: The von Neumann-Eckert Computer Model

Object-oriented (OO) Paradigm

An OO Program is a collection of objects that interact by passing messages that transform the state.

When studying OO, we learn about:

- Sending Messages
- Inheritance
- Polymorphism

Example OO languages:

- Smalltalk
- Java
- C++
- C#
- Python

Functional Paradigm

Functional programming models a computation as a collection of mathematical functions.

- Input = domain
- Output = range

Functional languages are characterized by:

- Functional composition
- Recursion

Example functional languages:

- Lisp
- Scheme
- ML
- Haskell

Logic Paradigm

Logic programming declares what outcome the program should accomplish, rather than how it should be accomplished.

When studying logic programming we see:

- Programs as sets of constraints on a problem
- Programs that achieve all possible solutions
- Programs that are nondeterministic

Example logic programming languages:

- Datalog
- Prolog

Special Topics

Concurrency:

e.g., Client-server programs

Correctness:

How can we prove that a program does what it is supposed to do under all circumstances?

On Language Design

Design Constraints:

- Computer architecture
- Technical setting
- Standards
- Legacy systems

On Language Design

Design Constraints:

- Computer architecture
- Technical setting
- Standards
- Legacy systems

Outcomes and Goals:

1. How does a programming language emerge and become successful?
2. What key characteristics make an ideal programming language?

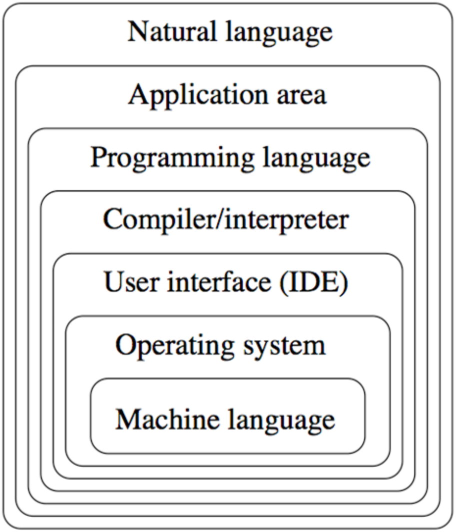
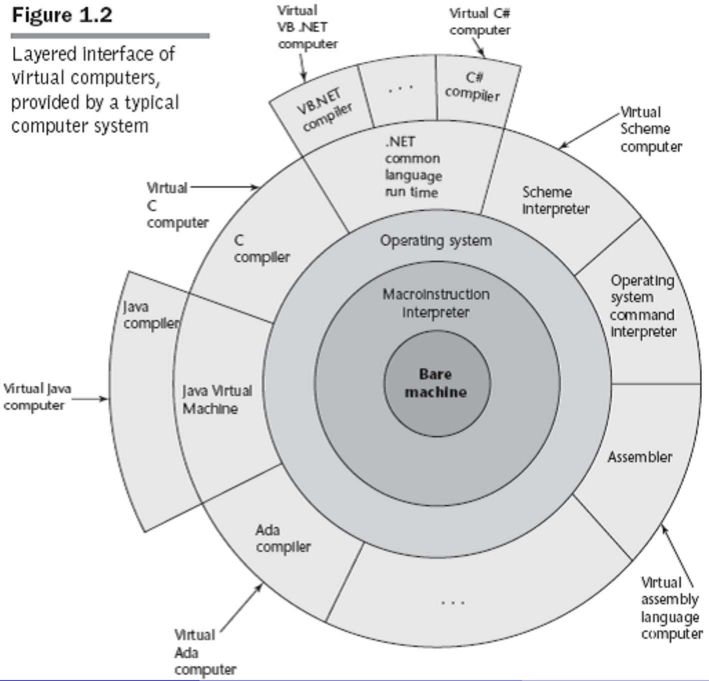


Figure 1.3: Levels of Abstraction in Computing

Figure 1.2

Layered interface of virtual computers, provided by a typical computer system



Language Design Trade-Offs

Reliability vs. cost of execution

- Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs

Readability vs. writability

- Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability

Writability (flexibility) vs. reliability

- Example: C++ pointers are powerful and very flexible but are unreliable

Compilation

Compiler – translate high-level program (source language) into machine code (machine language)

- Programs are translated into machine language; includes JIT systems
- Slow translation, fast execution
- Use: Large commercial applications

Compilation

Compiler – translate high-level program (source language) into machine code (machine language)

- Programs are translated into machine language; includes JIT systems
- Slow translation, fast execution
- Use: Large commercial applications

Example compiled languages:

- Fortran, Cobol, C, C++

Compilation

Compilation process has several phases:

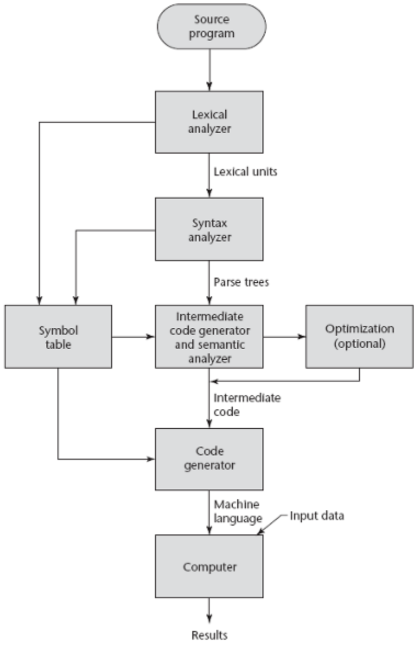
- lexical analysis: converts characters in the source program into lexical units
- syntax analysis: transforms lexical units into parse trees which represent the syntactic structure of program
- semantics analysis: generate intermediate code
- code generation: machine code is generated

Load module (executable image): the user and system code together

Linking and loading: the process of collecting system program units and linking them to a user program

Figure 1.3

The compilation process



Von Neumann Bottleneck

Connection speed between a computer's memory and its processor determines the speed of a computer

Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a bottleneck

Known as the **von Neumann bottleneck**; it is the primary limiting factor in the speed of computers

Pure Interpretation

Interpreter – executes instructions on a virtual machine

- Programs are interpreted by another program known as an interpreter
- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high-level languages but recent comeback with some Web scripting languages (e.g., JavaScript, PHP)
- Use: Small programs or when efficiency is not an issue

Pure Interpretation

Interpreter – executes instructions on a virtual machine

- Programs are interpreted by another program known as an interpreter
- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high-level languages but recent comeback with some Web scripting languages (e.g., JavaScript, PHP)
- Use: Small programs or when efficiency is not an issue

Example interpreted languages:

- Scheme, Haskell, Python

The Interpreting Process

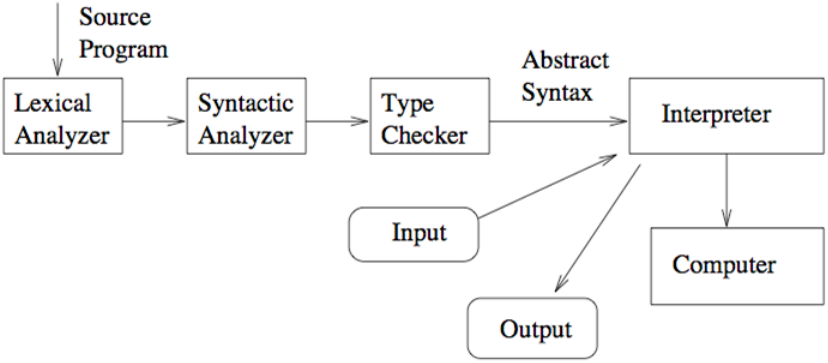


Figure 1.5: Virtual Machines and Interpreters

Hybrid Implementation Systems

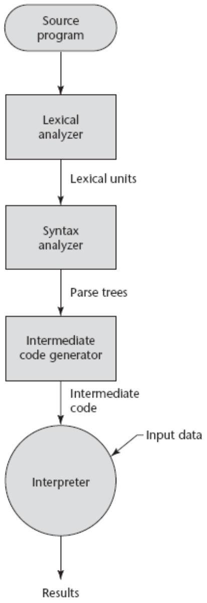
- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Use: Small and medium systems when efficiency is not the first concern

Example Hybrid compilation/interpretation

- The Java Virtual Machine (JVM)
- Perl programs are partially compiled to detect errors before interpretation

Figure 1.5

Hybrid implementation system



Just-in-Time Implementation Systems

Initially translate programs to an intermediate language

Then compile the intermediate language of the subprograms into machine code when they are called

Machine code version is kept for subsequent calls

JIT systems are widely used for Java programs

.NET languages are implemented with a JIT system

In essence, JIT systems are delayed compilers

Preprocessors

Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included

A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros

A well-known example: C preprocessor expands `#include`, `#define`, and similar macros

Characteristics of a Successful Language

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation

Characteristics of a Successful Language

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation

Simplicity and Readability

Small instruction set:

- e.g., Java vs Scheme

Simple syntax:

- e.g., C/C++/Java vs Python

Benefits:

- Ease of learning
- Ease of programming

Characteristics of a Successful Language

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation

Clarity about Binding

A language element is bound to a property at the time that property is defined for it.

Definition

A ***binding*** is the association between an object and a property of that object.

Clarity about Binding

A language element is bound to a property at the time that property is defined for it.

Definition

A ***binding*** is the association between an object and a property of that object.

Examples

- a variable and its type
- a variable and its value

Clarity about Binding

A language element is bound to a property at the time that property is defined for it.

Definition

A ***binding*** is the association between an object and a property of that object.

Examples

- a variable and its type
- a variable and its value

Early binding takes place at compile-time.

Late binding takes place at run time

Characteristics of a Successful Language

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation

Reliability

A language is *reliable* if:

- Program behavior is the same on different platforms

  e.g., early versions of Fortran

- Type errors are detected

  e.g., C vs Haskell

- Semantic errors are properly trapped

  e.g., C vs C++

- Memory leaks are prevented

  e.g., C vs Java

Characteristics of a Successful Language

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation

Language Support

Accessible (public domain) compilers/interpreters

Good texts and tutorials

Wide community of users

Integrated with development environments (IDEs)

Characteristics of a Successful Language

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation

Abstraction in Programming

Data:

- Programmer-defined types/classes
- Class libraries

Procedural:

- Programmer-defined functions
- Standard function libraries

Characteristics of a Successful Language

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation

Orthogonality

Definition

A language is ***orthogonal*** if its features are built upon a small, mutually independent set of primitive operations.

Orthogonality

Definition

A language is ***orthogonal*** if its features are built upon a small, mutually independent set of primitive operations.

Fewer exceptional rules = conceptual simplicity

Example

- restricting types of arguments to a function

Tradeoffs with efficiency

Characteristics of a Successful Language

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation

Efficient Implementation

Embedded systems

- Real-time responsiveness (e.g., navigation)
- Failures of early Ada implementations

Web applications

- Responsiveness to users (e.g., Google search)

Corporate database applications

- Efficient search and updating

AI applications

- Modeling human behaviors